Integrated Reproducibility with Self-describing Machine Learning Models

Joseph Wonsil
The University of British Columbia
Vancouver, British Columbia, Canada
jwonsil@student.ubc.ca

Jack Sullivan
Oracle Labs
Burlington, Massachusetts, USA
jack.t.sullivan@oracle.com

Margo Seltzer
The University of British Columbia
Vancouver, British Columbia, Canada
mseltzer@cs.ubc.ca

Adam Pocock
Oracle Labs
Burlington, Massachusetts, USA
adam.pocock@oracle.com

ABSTRACT

Researchers and data scientists frequently want to collaborate on machine learning models. However, in the presence of sharing and simultaneous experimentation, it is challenging both to determine if two models were trained identically and to reproduce precisely someone else's training process. We demonstrate how provenance collection that is tightly integrated into a machine learning library facilitates reproducibility. We present MERIT, a reproducibility system that leverages a robust configuration system and extensive provenance collection to exactly reproduce models, given only a model object. We integrate MERIT with Tribuo, an open-source Java-based machine learning library. Key features of this integrated reproducibility framework include controlling for sources of non-determinism in a multi-threaded environment and exposing the training differences between two models in a human-readable form. Our system allows simple reproduction of deployed Tribuo models without any additional information, ensuring data science research is reproducible. Our framework is open-source and available under an Apache 2.0 license.

CCS CONCEPTS

• **Information systems** → **Data provenance**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

Machine Learning, provenance, reproducibility

ACM Reference Format:
Joseph Wonsil, Jack Sullivan, Margo Seltzer, and Adam Pocock. 2023. Integrated Reproducibility with Self-describing Machine Learning Models. In *2023 ACM Conference on Reproducibility and Replicability (ACM REP '23), June 27–29, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3589806.3600039

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs International 4.0 License.

ACM REP '23, June 27–29, 2023, Santa Cruz, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0176-4/23/06.
https://doi.org/10.1145/3589806.3600039

1 INTRODUCTION

Reproducibility is the cornerstone of the scientific method. In recent years however, multiple disciplines, including machine learning and data science, have recognized a reproducibility crisis [3, 14, 33]. In ML, reliably reproducing trained models is a vital step in addressing the reproducibility crisis. We claim that *data provenance* is critical to reliable reproducibility. Data provenance is a formal record of the history of some digital object that documents how it came to be in its present form. The provenance for a machine learning model contains information about the training data, preprocessing, the training parameters, and the compute environment. Unfortunately, many existing systems add provenance collection post-hoc on top of an existing library. Such frameworks lack access to the internals of the system they track, preventing them from controlling potential sources of non-determinism that affect reproducibility. This approach requires proactive user effort, both when building and releasing models. As an alternative, we present Models Easily Reproduce In Tribuo (MERIT), which leverages provenance from the Tribuo ML library [28] to make it possible to exactly reproduce a model, given only the model object. The key enabler to MERIT is *integrated provenance collection*; the Tribuo library captures provenance internally with **no** user effort. Integrated provenance enables a reproducibility system to have direct access to library state, such as random number generators, which is critical for reproduction.

When discussing the reproducibility of ML systems, there are three general classes of "reproducibility" that we call: *bit-wise exact*, *execution-exact*, and *same recipe*. Bit-wise exact is the simplest to define, the original and reproduced models contain *exactly* the same floating point numbers, in the same order, down to their bitwise representation. Execution-exact means the same operations are executed according to the high level program, but two executions of that program might produce different results due to circumstances beyond the program's control (e.g., non-deterministic threading or different floating point order of operations). Same recipe is similar to execution-exact, except that the overall program flow may be slightly different (e.g., different approximations to the GeLU non-linearity, using TensorFlow instead of PyTorch). All three can be considered techniques to "reproduce" an ML model; we present a system that produces bit-wise exact reproducibility, with a fallback to execution-exact reproducibility in some cases (see Section 7 for details).

Our completely automatic reproducibility framework, MERIT, is embedded in Tribuo. This framework automatically re-trains a Tribuo model, given only its provenance or the model itself. It addresses and controls common issues that arise from non-determinism, particularly random number generator states and multi-threaded execution. Since Tribuo collects provenance automatically, users can share reproducible Tribuo models without advance planning or extra effort. They can also use the framework to re-train the model with a modified configuration and then generate a diff between the original and reproduced provenance to expose differences. These features provide a mechanism to audit the provenance of a model for regulatory compliance. Auditors can use MERIT to reproduce a model and check the reproduced provenance, weights, and evaluation to automatically verify whether the original provenance correctly described the lineage of the original model.

While developing MERIT we also identified gaps in the design of the Tribuo provenance system. Our investigation revealed provenance collection bugs that impeded the reproducibility of certain models and identified the need for new features, such as a method to directly modify RNG state after initialization. Additionally, our research uncovered reproducibility bugs in other libraries, such as hidden RNG state in Java ML libraries. By addressing these gaps, we not only added a reproducibility feature to Tribuo, but also improved the robustness of the entire framework and identified new areas of caution.

We make three main contributions. We present MERIT, a system that is integrated into an ML library that provides reproducibility without any a priori effort. We demonstrate how provenance integrated with the ML library is critical for enabling bit-wise exact reproducibility. We also provide guidance on how other ML libraries can be extended to facilitate such reproduction.

2 RELATED WORK

Reproducibility in machine learning (ML) model creation is a special case of the more general problem of computational reproducibility. The literature on general-purpose computational reproducibility is wide and varied [8, 23, 29, 30, 34, 35].

Despite these broad efforts in general computational reproducibility, ML reproducibility comes with its own unique set of challenges. A desired requirement for a reproducibility system is both to be able to re-train a particular model and to modify parameters of the model, tracking its evolution from execution to execution. Reproducibility systems must therefore collect ML-specific provenance such as hyperparameters, data transformations, and training data to achieve this goal.

The majority of related work is concerned primarily with how and where to collect this provenance. We note there are currently two primary approaches to ML provenance collection, differentiated by where they occur in the software stack. The most common is a 'wrapper' library that exists on top of existing ML libraries and records the state of the environment at each execution. Less common is a more general-purpose, fine-grain provenance collection at the level of individual operations occurring within a script. In the following sections, we describe the relevant work in each of these categories.

2.1 Wrapper Libraries

These frameworks provide wrappers around existing ML systems; each time a user trains a model, they snapshot the state of the experiment that uses the ML system. The data included in the snapshot state can differ depending on the framework and user input. However given that these libraries are ML-specific they usually collect information such as hyperparameters, RNG seeds, and code versions. Provenance collection systems that follow this approach include DeepDIVA [2], dagger [24], dtoolAI [16], and Schelter et al.'s automated metadata collection system [31]. While some of these systems are tailored to a specific library, others can wrap multiple. In summary these systems collect provenance for libraries including MXNet [7], SparkML [19], scikit-learn [26], and PyTorch [25].

The main method by which these systems support reproducibility is through reverting to saved checkpoints, Given that they operate on a user's environment, they also tend not to be portable. With the exception of dtoolAI, the provenance exists only within the training environment. Users wishing to reproduce each other's models will need to share environments in some way to make use of their features. With dtoolAI, the provenance is packaged with the PyTorch models it trains, but any other collaborator must be using dtoolAI as well since PyTorch cannot interpret the provenance.

Wrapper libraries have limitations imposed by their position in the software stack, particularly in regards to their lack of direct access to the state of the RNG. While they may record the seed used for a model, the use of multithreading or other changes in the order of operations can impede reproducibility of a model. Furthermore, these wrapper libraries must closely monitor each release of the machine learning (ML) libraries they support to ensure they can make necessary adjustments to capture provenance information for any new features. In contrast, Tribuo's provenance collection system integrated directly into the library evolves alongside new features, utilizing developer-defined APIs and compile-time checks to facilitate the process.

2.2 Fine-grain Collection

Systems that collect fine-grain provenance on data science scripts often have different or broader goals than reproducibility. For example, NoWorkflow [22] and RDataTracker [18] collect fine-grain provenance at the source code level for Python and R respectively, as well as environmental information. Typically, this results in provenance that is a superset of that captured by wrapper libraries, since they are more general-purpose and not necessarily ML-specific.

On the other hand, LIMA [27] is an integrated provenance collection for SystemDS [4] that records fine-grain provenance during training to provide automatic memoization, similar to IncPy [15]. While its goal differs from that of Tribuo, its level of detail means that its lineage logs can theoretically assist in debugging deployed models and reproducing a model from scratch. Additionally, as an integrated system, LIMA circumvents many of the problems found in wrapper libraries.

Despite the benefits these libraries provide, they also have drawbacks. In particular, RDataTracker and NoWorkflow impose performance penalties due to the overhead of fine-grain collection. This degradation can be limited by controlling the levels of provenance collection; however, certain workloads can still be dramatically

affected. LIMA improves performance, but it collects a high volume of data at a level far deeper into the software stack than the abstraction of the machine learning pipeline. This volume and granularity of data is ill-suited for bundling with a model for the purpose of reproducibility.

Tribuo has integrated provenance such as LIMA; however, it is tailored explicitly for tracking the creation of an ML model and its evaluation. This granularity imposes minimal overhead while still providing the necessary information for reproducibility and auditing of an ML model.

3 TRIBUO OVERVIEW

Tribuo [28] is an open source Java Machine Learning library, released publicly in 2020 under an Apache 2.0 license. It is designed for compile-time, type-safe ML computation, and the models, datasets and evaluations produced via Tribuo are *self-describing* through the incorporation of provenance. We first discuss Tribuo provenance and the implications it has on reproducibility. For a more detailed description of Tribuo, see Pocock [28].

Tribuo's provenance system tracks the creation of each Tribuo core object: `DataSource`, `Dataset`, `Trainer`, `Model`, and `Evaluation`. Each of these objects contains a `Provenance` object that records the computation that led to the construction of the host object. Provenance objects contain the type and version of their host object along with object-specific information. `DataSource` provenance objects store the file path and file hash, and either a DB query, the parameters of the data generator, or the feature extraction process to allow reconstruction of the `DataSource`. `Dataset` provenance objects store a reference to the provenance of the data source, along with provenance describing any transformations that have been applied to the data. `Trainer` provenance objects store the training hyperparameters along with sufficient information to recover the RNG state if the training algorithm requires a source of randomness. `Model` provenance objects store the training dataset provenance and the trainer provenance, along with instance level information such as the Tribuo version, OS, CPU architecture and JVM version, along with optional user-supplied metadata about the specific run that created the model. Appendix F contains a complete provenance object of a `Model`. Finally, `Evaluation` provenance objects contain the model provenance for the model being evaluated and the evaluation dataset or datasource provenance. Note that multiple objects can contain references to the same provenance object, and as the objects are immutable, this is equivalent to, but more efficient than, copying the provenance into each object.

These provenance objects are what make Tribuo models and evaluations self-describing. Each model knows how it was created, its training parameters, training data pipeline, and metadata such as numbers of features, samples, and output dimensions. Provenance collection was designed to track models in production, removing the possibility of any skew between the model metadata tracking system and the model itself, because the model is the source of truth for its own construction. Additionally the storage of the training data pipeline allows models to be loaded from disk and have their input pipeline reconstructed automatically, including information such as which columns provide which features.

Tribuo runs on the JVM, which is an inherently concurrent and parallel system, unlike the single-threaded reference implementation of Python used by most ML libraries. This means that objects in Tribuo such as `Trainer` and `Model` can be accessed concurrently by multiple threads, and any shared state such as a global RNG can also be accessed concurrently. Therefore shared mutable state such as a global RNG or the RNG inside a `Trainer` must be monitored by the provenance system to ensure the state is properly tracked. For this reason, Tribuo adopts Java's `SplittableRandom` RNG [32] as its default RNG and disallows any global RNGs. The RNG state is tracked by counting how many times each RNG has been invoked, where an invocation involves splitting a training-run-specific RNG from the top level RNG inside a `Trainer` during each call to `Trainer.train`. Each training run can take an unknown number of draws from the RNG, which can be dataset size dependent (e.g., shuffling the examples before an epoch of stochastic gradient descent), model dependent (e.g., building a Random Forest, where features are subsampled at each leaf in each tree) or both. Using a local RNG split from the host one ensures that the state of the host RNG is correctly tracked by the provenance counter, while still providing an independent source of randomness for each training computation. These RNGs are critical for capturing state necessary for reproducibility that is otherwise lost in ML pipelines.

Finally, correct and complete provenance collection requires careful control of the order of operations when working with floating point numbers. Non-deterministic scheduling can cause a different order of operations in reduction computations such as dot products, leading to identical code producing different learned parameters. Many of Tribuo's training procedures are sequential so there is no variation of the order of operations (as the floating point behavior is guaranteed by the Java Language Specification [13]). For multi-threaded computations, Tribuo uses the Fork-Join framework, which can guarantee a reproducible order of the reductions with careful task construction.

Tribuo's provenance collection has some limitations. It stops at the language and class library border, recording the JVM version and system architecture in its provenance. Optimizations within the JVM (e.g., due to the JIT compiler) that change the output are not tracked. The Tribuo design philosophy also limits flexibility by retaining control of the training loop in exchange for provenance collection with insignificant overhead. We discuss this more in Section 7.

4 REPRODUCIBILITY CHALLENGES

We identify a set of challenges that a framework must address to reproduce an ML model. Many of the tools described in Section 2 identify and address some of these issues, but to the best of our knowledge, MERIT is the only system that automatically addresses all of them through direct integration with an ML library.

C.1 Obtaining a complete record of a model's provenance.
C.2 Re-instantiating correctly configured objects.
C.3 Addressing non-determinism due to RNGs and parallelism.

Challenge **C.1** is particularly important since without a record of the process one is trying to reproduce, reproduction is impossible. Many algorithms, transformations, hyperparameters, and optimizations combine to produce an ML model. In most systems, users must

know they need provenance collection and take explicit action to capture it, since most ML systems do not capture provenance automatically. Then, they must choose a framework to provide this functionality, confirm that it is compatible, and integrate the framework into their workflow.

The underlying question is, "At what layer can we record the execution of a process (i.e., obtain its provenance)?" Previous work [2, 16, 22, 24, 31] collected provenance in a layer on top of an existing system during model training and hyperparameter optimization. This approach creates a trade-off between instrumentation and automation. Frameworks such as dagger [24] require significant instrumentation but allow the user to specify precisely what is collected. On the other hand, noWorkflow [22] requires zero instrumentation but introduces potentially significant performance overhead and leaves the user responsible for parsing the resulting provenance to extract the information needed. A framework with provenance collection built-in from the start avoids these trade-offs, as discussed in Section 3.

Once we have provenance, converting it into something useful to an ML system is a second challenge, **C.2**. Provenance often appears in language-independent formats such as JSON or XML. While interoperable, portable, and readable, these formats may be lossy compared to direct object serialization. Further, while provenance is immutable, the code it describes changes over time. As software evolves, reproducing models becomes more challenging as the underlying behavior of the training system differs from version to version. For example, APIs change and deprecate; in September 2021, scikit-learn [26] deprecated the use of `np.matrix`[1]. To reproduce a model created before this change, a user must roll back their version of scikit-learn to an earlier one with the deprecated function. Another problem can arise when hardware changes; floating-point arithmetic can change, so a record of host architecture is often helpful. We discuss how Tribuo addresses these problems in Section 3 and how MERIT uses this information in Section 5.3.

Finally, provenance collection in conjunction with a reproducibility framework must identify and compensate for all sources of non-determinism, Challenge **C.3**. The most obvious source of non-determinism is the RNG(s). Collecting RNG seeds is necessary but not sufficient. RNGs construct their initial state from a seed, but this internal state changes with each method invocation. Since users might train multiple models using the same RNG, that seed is only sufficient for training the first model. Training a specific model requires either re-training all models (likely a costly undertaking) or recording the RNG seed *and* starting state for each model. No provenance recording system of which we are aware, aside from Tribuo, captures such information. Furthermore, even if the provenance contained this RNG state, the ML library must provide a way to adjust the RNG to the appropriate state, a functionality missing in current systems. We discuss how we address this problem with MERIT in Sections 5.4 & 5.5.

[1] scikit-learn.org/dev/whats_new/v1.0.html

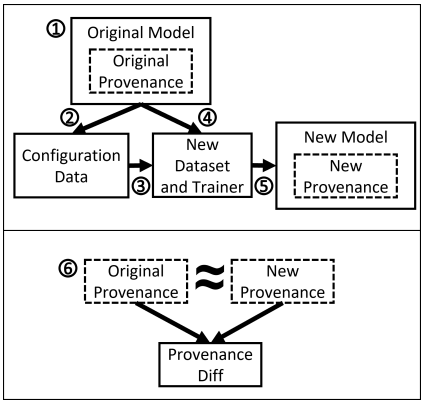

Figure 1: Overview of the reproducibility system.

5 INTEGRATED REPRODUCIBILITY DESIGN

MERIT builds on Tribuo's self-describing models, fine-grain control of random number generators (RNGs), and the Oracle Labs Configuration and Utilities Toolkit (OLCUT) configuration system[2] to provide integrated reproducibility. Using these components, along with extensions we made to Tribuo, we address the challenges described in Section 4. The result is a fully integrated reproducibility system for any Tribuo-trained model and a tool to compute diffs between two models' provenance. We first provide an overview of our work and then discuss the major components in the following subsections.

5.1 Overview

`ReproUtil` is a new Java class that provides the API to MERIT. It takes either a Tribuo model object or a model provenance object as input (since every model object contains a provenance object) and produces a new model, trained identically to the original, as an output. Figure 1 illustrates this process.

Since the provenance is a superset of the configuration used to train a model, MERIT extracts all configuration data from the provenance **(2)** and imports it into an OLCUT configuration manager. The configuration manager can parse the data to re-instantiate any configurable object described in the provenance **(3)**. This process recreates many of the Tribuo objects necessary to reproduce a model, MERIT instantiates the rest via reflection.

Next, we address the challenge of reproducible RNGs. Each trainer object in Tribuo has an RNG that must be initialized correctly. Tribuo provenance stores RNG seeds, but the internal state of the RNG changes each time it is used, so a Java program that trains multiple models has only a single seed but different initial states for each model. Tribuo trainers also record an *invocation count*, the number of times they use an RNG, and we use this count to set the RNG's state correctly for a given model **(4)**.

At this point, all objects Tribuo used to train the original model are re-instantiated and set to their correct state. MERIT splits the data into training and testing as necessary and then trains a new model **(5)**. Once a user has this new model, MERIT's diff tool **(6)** compares the provenance of the new and original models. If both

[2] github.com/oracle/olcut

models were trained on the same hardware without any external (to Tribuo) form of non-determinism, the diff will show that only the timestamps of the models differ.

We next present each component of MERIT. First, we describe its usage in terms of inputs and outputs. Then, we discuss how we use the provenance to re-instantiate Tribuo objects. Next, we explain how we set the RNG using the seed *and* the invocation count. Ensemble training further complicates the RNG state, so we address that issue next. Finally, we discuss our diff tool and how we use it to verify reproduction and identify bugs.

5.2 Usage

Given a provenance object, `ReproUtil` executes the reproducibility pipeline. When the pipeline completes it can compare metadata that appears in the original and reproduced models, e.g., the feature and output domains. If these feature and output domains differ, then either the data file changed but produced the same hash (a rare, but possible event), feature names changed, or some other change, unobservable by the provenance system, occurred (e.g., a system library was updated). This check can both expose bugs within the reproducibility framework and provenance system and confirm that the developer provided the correct data. Finally, assuming the `ReproUtil` instance threw no exceptions, it returns a new model. This new model is the same type as the original, and MERIT trained it under the same conditions.

5.3 Object Re-instantiation

Most Tribuo objects are configurable through the OLCUT configuration system. OLCUT instantiates components from configuration files on the fly using dependency injection, allowing specification of objects in a readable format. The provenance collection system is heavily integrated with OLCUT, as any provenance object in Tribuo inherits its type from OLCUT, ensuring that the object's configuration is recorded in the provenance. This integration means we can use the `ProvenanceUtil` from the OLCUT library to extract the configuration from provenance objects, which returns a list of `ConfigurationData` objects that we add to an OLCUT `ConfigurationManager`.

When a user invokes the `reproduce` method on their `ReproUtil` instance, it retrieves new instances of the objects from the new `ConfigurationManager`. In this way, we quickly create correctly configured `Trainer` and `DataSource` objects. The only objects we cannot instantiate this way are the `Dataset` and, if the model used one, a `TrainTestSplitter`. `Datasets` are not configurable, so instead the `ReproUtil` object examines the provenance for the class name of the `Dataset` and instantiates it using reflection. If the model used a `TrainTestSplitter`, we search the provenance for its split ratio and RNG seed, allowing us to instantiate a new identical instance. Together, these methods instantiate all the objects necessary to reproduce a given model.

5.4 Setting RNG State

We must set the RNGs of Tribuo `Trainers` to the state they were in when Tribuo originally trained the model. Each time a user calls `train`, the corresponding `Trainer` splits its RNG and saves the result as a `localRNG` used for all random number generation in that `train` call. If a developer trained a sequence of models $\{S_0 ....S_n\}$ using the same `Trainer`, and they want to reproduce the S_n^{th} model, they need to set the RNG to its state after it trained the S_{n-1}^{th} model. Given Tribuo's usage of RNGs in the `train` method, the correct RNG state can be reached by re-instantiating the RNG and splitting it $n-1$ times.

As explained in Section 5.1, Tribuo's provenance system provides us with the necessary information to identify the RNG state: the seed and the invocation count. Tribuo protects this count from multithreading problems using Java's `synchronized` keyword. Each `Trainer` 'synchronizes' on itself for a block of code, meaning that out of all the threads that hold a reference to that `Trainer`, only one can execute in the block at a time. When a user invokes a `Trainer`'s `train` method, the `Trainer` enters the `synchronized` block, records its provenance, splits the RNG, increments the invocation count, and exits the `synchronized` block. This synchronization guarantees that if two threads hold a reference to the same `Trainer` object, and they both call train simultaneously, there is no race condition where they receive different `localRNGs` but record the same invocation count. This process also ensures that the S_n^{th} model trained by a `Trainer` has an invocation count recorded in its provenance equal to $n-1$.

Tribuo begins creating an RNG using the seed from the provenance, which produces an RNG with an invocation count of zero. Unfortunately, the only way to modify the internal RNG state is through calling the `train` method. The RNG cannot be accessed directly as it is a private field, and since its internal state is a function of its use, we cannot instantiate it to a specific state. Training a model $n-1$ times just to change the RNG state adds significant overhead, so we altered Tribuo's `Trainer` interface. We added a new method, `setInvocationCount`, that takes a positive integer i as input, recreates the RNG, and then splits the RNG i times.

This new method combined with the existing provenance allows a `ReproUtil` object to set the RNG state of the `Trainer` that it will use to reproduce a model. `ReproUtil` searches a model's provenance for any `Trainer` provenance objects, collects the recorded invocation counts, identifies the corresponding re-instantiated `Trainer` object, and sets the invocation count accordingly.

5.5 Ensemble Trainer Non-determinism

Ensemble trainers are not immediately compatible with our reproducibility framework due to issues when tracking RNG state in concurrent programs. As a running example, we discuss the Tribuo `BaggingTrainer` and the issues created by multithreading non-determinism. We then explain how we modify the `BaggingTrainer` to ensure that all the information necessary to reproduce it is in its provenance, thus requiring no special-casing in the framework.

Consider Tribuo's Random Forest algorithm, which extends the `BaggingTrainer`. Users instantiate a `RandomForestTrainer` and provide it with an `innerTrainer` that trains each decision tree in the Random Forest. The user can retain a reference to the `innerTrainer` and train another model in parallel while the Random Forest algorithm executes. Each time a thread calls the `train` method, the trainer will briefly synchronize on itself to split the RNG and collect provenance as discussed in Section 5.4. This synchronization ensures that it trains each model with a unique state of

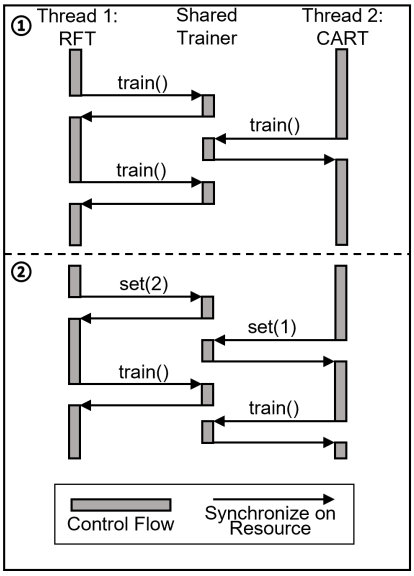

Figure 2: Example problematic scenarios in which parallel computations share the same trainer (CART). In one thread the CART is an inner trainer to a RandomForestTrainer (RFT), the other holds a direct reference. The synchronization on the trainer is only for a brief period at the start of training, not the entire train() invocation.

the RNG. However, since two algorithms might execute in parallel, the distribution of RNG states between the two is non-deterministic. This behavior is not a problem for two threads in the same Random Forest but can cause reproducibility issues when some threads are training a Random Forest *and* other threads are training a separate decision tree with the same trainer.

For example, Figure 2 **(1)** shows how a Random Forest training in one thread can result in non-contiguous RNG states, because another thread uses the same trainer. When re-training this Random Forest, if that extra call to `train` is not invoked or simulated, the resulting model will differ from the original since it changes the state of the RNG for subsequent invocations. In other words, you cannot get the same set of models by splitting the RNG once per invocation of the `train` method. All the information necessary to reproduce this model resides within the provenance; however, any attempt at automation requires that the framework be aware that the trainer is a Random Forest, deconstruct the trainer, train each decision tree with the correct RNG state, and reconstruct everything into the correct model. This approach is likely how a developer might manually reproduce a model, but in our framework, re-training in this manner can encounter the initial multithreaded problem.

One of the potential benefits of the `BaggingTrainer` is precisely its ability to train models in parallel. If we were to re-train this model in parallel after carefully setting the invocation count of the inner trainer, we would encounter a race condition where one thread sets the invocation count of the inner trainer, but another invokes `train` first, Figure 2 **(2)**. We could sidestep this problem by training each inner model sequentially rather than in parallel, but we would then lose the performance benefit of parallelism. Instead, we avoid special cases and losing parallelism. We modified the `BaggingTrainer`'s algorithm and extended the `Trainer` interface to allow our reproducibility framework control over the RNG each time it invokes `train`.

We overloaded the `train` method to add a new parameter, an invocation count. This change allows us to set the invocation count within the synchronized block during training, eliminating any race conditions with the RNG. Passing an invocation count to any call to `train` guarantees that the resulting model used that state of the RNG for its training, regardless of any threads referencing the same object.

We use this new method to ensure that the invocation counts used in the `BaggingTrainer` are contiguous. As a result, when a `BaggingTrainer` begins to train, it instantiates an invocation tracker that we initialize to the inner trainer's current invocation count. Each time it trains a single model, it passes the value of the tracker to the `train` method and then increments the tracker. This guarantees that if an inner trainer has an invocation count of i, and there are N ensemble members, then regardless of the order in which they were trained, the set of invocation counts from the resulting models will be $\{i, i+1, ..., i+N\}$. Even if another thread holds a reference to the inner `Trainer` and calls `train` in parallel to the `BaggingTrainer`, the inner models will have contiguous invocation counts since the `Trainer` sets the RNG state in a synchronized block. A consequence of this choice is that it is possible for a thread using the same `Trainer` in parallel to train a model using the same RNG state as one of the models in the `BaggingTrainer`. However, since the randomness across models in the `BaggingTrainer` is preserved, reproducible, and deterministic, we believe this to be acceptable behavior.

These changes to the `BaggingTrainer` ensure that the only information necessary to reproduce its model is the RNG seed, the `BaggingTrainer` invocation count, and the inner `Trainer` initial invocation count, all of which are in the provenance. Whenever a user invokes the reproduce method on a `ReproUtil` instance, it searches the provenance for *all* `Trainer` provenance and sets *all* corresponding `Trainer`'s invocation counts before re-training. Even though we cannot access an inner `Trainer` through the `BaggingTrainer` interface, by matching provenance objects to their corresponding re-instantiated versions held by the configuration manager, we can set all `Trainer` invocation counts correctly. While we used the `BaggingTrainer` as an example here, we can apply this methodology to other ensemble models in Tribuo. It is worth noting that concurrently using an inner `Trainer` with an ensemble `Trainer` that contains it is not an idiomatic use of Tribuo. However, as we need accurate provenance in all circumstances, and the JVM is a concurrent & parallel platform, then this kind of defense is necessary.

5.6 Provenance Diffs

The final feature of our reproducibility framework is a utility that creates a diff between two models' provenance. We chose to write a diff tool rather than implement an 'equals' method because we know that a reproduction's provenance will *not* equal the original's

provenance. *This is not a result of poor reproduction; it is due to the fact that provenance represents a single execution.* Some values are guaranteed to be different from execution to execution, e.g., timestamps. Additionally, users may want to use the reproducibility system as a basis for optimizing hyperparameters. In this scenario, a diff is an invaluable tool for comparisons between models as it highlights differences in optimizers, number of epochs, the RNG state, and any other configuration options that might have changed.

The diff tool takes two models' provenance objects and creates a JSON string that contains the differences (a *report*). Each key in the report is the name of a provenance object or a model designation. The values in the report are more keys or the value of *primitive provenance*. Primitive provenance is a provenance object that references no other provenance objects, such as a `String` representing the name of a `Class`. We list the possible types of primitive provenance in Appendix A. Currently, the tool designates an 'original' and a 'reproduced' model. When adding the value of a primitive provenance object to the report, the tool places the value under the keys 'original' or 'reproduced.' However, this mechanism is designed to be easily extensible in case the tool is diffing models that do not fit the relationship of "original" and "reproduced".

For all objects contained in *both* models, the tool recurses until it reaches primitive provenance. It compares the two values and, if they are *different*, adds the comparison to the resulting report nested in its correct position. For example, the timestamp for the creation of a `DataSource` is found nested within the keys of the provenance objects that contain it, as seen in Listing 1. If one object is not in the other, we add the entire object tree to the diff.

Listing 1: An example diff between a model's provenance and the reproduced model's provenance. We have truncated the timestamps for space.

```
{
  "dataset" : {
    "datasource" : {
      "source" : {
        "datasource-creation-time" : {
          "original" :
          "2021-09-30T19:54:01.99",
          "reproduced" :
          "2021-09-30T19:54:03.83"
        }
      }
    }
  },
  "trained-at" : {
    "original" :
    "2021-09-30T19:54:02.09",
    "reproduced" :
    "2021-09-30T19:54:03.83"
  }
}
```

Based on our chosen behavior, the reports generated by our diff tool tend to be smaller the more similar the two models are. For example, a diff between an original model and a reproduced one created by MERIT on the same machine differ only in timestamps, Listing 1. This result indicates that Tribuo trained these two models with identical pipelines.

MERIT's diffs also avoid blow-ups and redundancy when it can recognize that different objects have similar functionality, assuming they follow a standard naming convention, typically enforced through inheritance. For example, Listing 2 shows the diff between two models trained using tree ensemble trainers. Tribuo's `RandomForestTrainer` and `ExtraTreesTrainer` are subclasses of the `BaggingTrainer`, with the restriction that the inner trainer is a tree trainer in both cases. Despite being different classes, the diff has correctly recursed into the fields of the ensemble trainers, determined that the tree trainer is the same across both classes, and exposed only the difference in the tree trainer parameters. Individually these provenance objects are each 2000 lines [3]. We provide more example diffs in Appendix C.

Listing 2: An example diff of two tree ensembles, one RandomForest and one extremely randomized tree ensemble. Both provenance objects pre-diff are 2000 lines. We have removed the timestamps and shortened class names for clarity.

```
{ "trainer" : {
    "class-name" : {
      "original" : "...RandomForestTrainer",
      "reproduced" : "...ExtraTreesTrainer"
    },
    "innerTrainer" : {
      "fractionFeaturesInSplit" : {
        "original" : "0.5",
        "reproduced" : "1.0"
      },
      "useRandomSplitPoints" : {
        "original" : "false",
        "reproduced" : "true"
      }
    }
  }
}
```

6 RESULTS

There are various methods to evaluate whether or not a Tribuo model is the 'same' as another. We choose to demonstrate the effectiveness of MERIT by reproducing a variety of models, examining the provenance diffs, comparing the model evaluations, and comparing model weights. These metrics allow us to test for *execution-exact* reproduction and *bit-wise exact* reproduction, as defined in Section 1.

6.1 Methodology

We identify two metrics for a successful *execution-exact* reproduction; 1) values in a provenance diff and 2) comparisons of the evaluations. We previously discussed in Section 5.6 how provenance diffs

[3]We uploaded the original provenance objects to our artifact at https://doi.org/10.5281/zenodo.7987883

containing only timestamps identify two models that Tribuo trained with the same pipeline. In addition, Tribuo has `Evaluator` classes that perform standard ML model evaluations. For classifications, it computes accuracy, precision, recall, F-score, and the confusion matrix. For regressions, it calculates the root-mean-square error, mean absolute error, and R^2 metrics. While two models might differ in their predictions but have similar provenance, it is much less likely that different models will have the same evaluation metrics.

We use these two metrics to define an *execution-exact reproduction* as an instance where two models differ only in timestamps related to start and end times, and the evaluations of the models are identical. For *bit-wise* reproduction we use these two metrics and a direct comparison of float values in a model's weights.

Next, we need tasks and data that we can use to train models for these metrics. We use Tribuo's tutorials[4] and example `Trainer` configuration files to generate example models. The tutorials demonstrate how to perform common ML tasks in Tribuo and explain Tribuo's features. They are a set of Jupyter notebooks that cover various tasks: classification, regression, and clustering, and a wide range of data sources: including IDX, CSV, JSON, directories of text files, and data generators. They use well-known datasets in the ML literature: the irises dataset [12], MNIST [17], wine quality [9], yeast [10], and Twenty Newsgroups [20]. Some models transform data using feature scaling or splitting between train and test data, and some train without any transformations. Meanwhile, the example `Trainer` configurations help ensure our reproducibility framework is successful for various algorithm types. Together, these factors help ensure that our reproducibility framework works with commonly used ML tasks and data.

Our process of reproducing and evaluating models is straightforward. First, we need models to reproduce. For the models from the tutorials, we directly copy the necessary code to load the dataset and train the model into a new notebook. Unlike the tutorials, the configurations do not have corresponding datasets. Fortunately, the configurations are categorized by task, so we loaded one dataset for each task: irises for classification, yeast for multilabel classification, and wine quality for regression. Then we build an instance of each trainer from the configuration and train a model.

We test reproducing models on the same machine and on different machines with varying hardware and architectures. To facilitate this process, whenever we train one of our original models we also serialize it to a file. Then, when on different machines, we can deserialize the original models for re-training on different hardware.

Once we have trained models, we can reproduce them and make comparisons. We pass the model as input to our reproducibility framework and invoke its reproduce method, which is two lines of code. Once we have both models, we diff their provenance to see if any values changed other than expected timestamps. Then we use an `Evaluator` to produce an evaluation for each and compare them. When comparing across machines we also verify bit-wise reproducibility for models with weight matrices. Finally, we record the provenance, diff, and evaluation comparison.

Table 1: Our evaluation reproduced models using this set of tasks and algorithms. Some rows correspond to one model, while others correspond to multiple models trained with different hyperparameters.

Task	Algorithm
Anomaly Det.	One-class RBF-SVM
Classification	Bagging Ensemble
Classification	CART Classifier
Classification	Extremely Randomized Trees
Classification	Factorization Machine
Classification	K-Nearest Neighbor
Classification	LibLinear Classifier
Classification	LibSVM RBF-SVM
Classification	Logistic Regression/AdaGrad
Classification	Multi-class Adaboost
Classification	Multinomial Naive Bayes
Classification	Random Forest
Classification	XGBoost Classifier
Clustering	K-Means
Multi-label	Classifier Chain
Multi-label	Classifier Chain Ensemble
Multi-label	Factorization Machine
Multi-label	K-Nearest Neighbor
Multi-label	Logistic Regression/AdaGrad
Regression	CART Joint Regression
Regression	CART Regression
Regression	ElasticNet/Coordinate Descent
Regression	Extremely Randomized Trees
Regression	Factorization Machine
Regression	K-Nearest Neighbor
Regression	Least Angle Regression
Regression	LibLinear Regression
Regression	LibSVM RBF-SVM
Regression	Linear Regression/SGD
Regression	Linear Regression/AdaGrad
Regression	Random Forest
Regression	XGBoost Regression

Table 2: Datasets and their corresponding format we evaluated.

Dataset	Data Format
Generated	Generator
Generated	CSV
Generated	JSON
Irises	CSV
MNIST	IDX
Twenty Newsgroups	Text files - Unigram tokens
Twenty Newsgroups	Text files - Bigram tokens
Wine Quality	CSV
Yeast	LibSVM

6.2 Reproduction Results

We trained 48 models, 14 from the Tribuo tutorials section and 34 from the example configurations. For reproducing on a single machine, Table 1 presents the various tasks and algorithms we tested, and Table 2 presents the datasets we used. These tables display aggregated information since some models were the same with varying hyperparameters or input data. We were able to successfully reproduce all of the models we trained[5]. While we cannot show all of the diffs due to size constraints, they contained only timestamps, and the evaluations produced the same values. Given the breadth of tasks and algorithms we used, we believe this result shows our framework is robust to common use cases that Tribuo supports[6].

For reproducing on different machines, we used one of each algorithm from the Regression and Classification tasks on the wine and irises datasets respectively. We chose a simple subset to evaluate rather than all the previous experiments since it takes only one failure to achieve bit-wise reproducibility for us to relax our reproducibility claim to recipe-exact instead. We trained them on x86 and reproduced them on another x86 machine and an ARM machine. Environment details are in Appendix D. We achieved bit-wise reproducibility between x86 machines. We also achieved execution-exact reproducibility between x86 and ARM based on the metrics in Section 6.1; however, we did not achieve bit-wise reproducibility due to differences in the JVM, as we found weight differences up to 6.4×10^{-16}. We discuss the reasons for this difference next in Section 7 and a more in-depth discussion on cross-architecture reproduction is in Appendix E.

While we do not explicitly evaluate the full performance of Tribuo's provenance collection system, we note that the tradeoffs in its design lead to low overhead. For example, each model in Tribuo's random forest implementation with ten trees takes less than or equal to one millisecond to capture provenance on the wine quality dataset. In these experiments the total training time took 180 to 130 milliseconds depending on JIT compiler optimizations. This cost is roughly constant as it usually will not scale with dataset size. An exception is a `DataSource` that uses a `RowProcessor` which will create an object per feature in the dataset. However, the cost is still minimal as the creation of each object is cheap, especially when compared to the time of the training loop.

7 DISCUSSION AND LIMITATIONS

Tribuo's integrated provenance collection via self-describing models was a significant step towards reproducibility and laid the groundwork for MERIT. Provenance collection provides useful information about models; however, without a well defined use case such as reproducibility it is difficult to tell if the provenance collection is sufficient to describe the computation or completely captures all the sources of variability. Building MERIT exposed deficiencies in Tribuo's provenance collection that we fixed.

7.1 Tribuo and Ecosystem Improvements

We used the diff tool to identify bugs in the provenance collection implementation in some of Tribuo's `Trainers`. Typically we caught these bugs, because MERIT threw an unexpected error or unexpectedly failed to produce a bit-wise exact reproduction. Occasionally, during implementation of MERIT, we uncovered non-fatal errors.

We identified an issue with multidimensional regressions where a mapping between dimension names and ids broke due to reliance on the order of an unsorted map. We also found two trainers with bugs where the recorded RNG state could be incorrect. Outside of training, we discoverd a data source where an equality check on a filepath could fail since only one side would normalize a path containing either "`.`" or "`..`". This issue is particularly interesting, as someone reproducing a model manually might not catch it, because they would see that the two file paths were the same. We also discovered the `CSVDataSource` class was not completely integrated with the provenance collection and required refactoring. We successfully implemented fixes for all of these issues, and they are now merged into the main Tribuo code.

Outside of Tribuo, our development led to the discovery of hidden RNG state in the `LIBLINEAR` [11] and `LIBSVM` [5] Java libraries. We reported these issues, and in the case of `LIBLINEAR`, the developers implemented a change to address the un-captured global RNG state we found[7]. Future bugs in either Tribuo or a user's model might also be caught thanks to the ability of the reproducibility framework to control many of the variables that otherwise might create interference when debugging.

As we discuss in Section 6.2, changes between CPU architectures can cause a reproduction failure, which led to a new provenance feature: recording the host architecture and JVM version. We noticed that Tribuo's logistic regression training algorithm produces slightly different models when trained on x86_64 and ARM64 architectures. We traced this to a difference in the implementation of the `Math.exp` compiler intrinsic between the two architectures, and the CPU architecture and JVM version difference is now noted in the provenance diff (see Section 5.6), but not elsewhere in the provenance. If Tribuo's provenance is integrated with system level provenance [21], then we could further track differences in the environment that can impact the computation. A more detailed explanation of this behavior is in Appendix E.

7.2 Limitations

Our reproducibility system has limitations in terms of both data collection and reproduction fidelity. MERIT operates on Tribuo-trained models of the same version since they take advantage of the provenance collection. Therefore, we lose reproducibility guarantees when Tribuo loads external models saved in TensorFlow [1] or XGBoost [6] formats and different versions of Tribuo. We cannot reproduce all models prior to this version, 4.2.0. Future versions will have backwards compatibility; however, they might be execution-exact rather than bit-wise due to optimizations changing the underlying computations. Although the version change will appear in the provenance diff to help explain potential model differences. Additionally Tribuo's deep learning implementation

[4] github.com/oracle/tribuo/tree/main/tutorials

[5] Tribuo has a TensorFlow trainer and tutorial, which we did not attempt to reproduce. We discuss this in Section 7.

[6] This evaluation is available for reproduction using our uploaded artifact or at https://doi.org/10.5281/zenodo.7987883

[7] https://github.com/bwaldvogel/liblinear-java/issues/40

relies on TensorFlow-Java, which, at submission time, has known reproducibility bugs, so we do not consider it further[8].

An intentional limitation on MERIT is its lack of ability to *automatically* verify reproductions. Since we also created MERIT to provide a way to make controlled changes to models, we did not want to automatically flag different models since a different model might be the goal for some users. Future work could involve extending the type system of Tribuo to include "variable" types with relaxed equality so types such as timestamps would not cause comparing diffs to fail an automated check.

MERIT assumes that all provenance collection implementations are correct. However, Tribuo is an extensible framework using object-oriented design principles, so developers can implement new trainers, data sources, or other objects. Interfaces ensure that these objects have provenance collection methods, but cannot ensure that the implementations are correct.

Tribuo also provides a feature for removing provenance from a model before deployment, in case the provenance contains sensitive information. Although the provenance object is removed, it is replaced by a hash of its contents. In this case, our framework cannot use the model itself since they are no longer self-describing. However, if the user saved the provenance before removal, they can still reproduce the model using the saved provenance.

Additionally there are events outside of Tribuo's control, such as user overrides of class paths and library versions, JVM optimizations, external libraries such as `LIBLINEAR`, and differences between hardware architectures. Zhuang et al. specifically identified software choice and hardware architecture as typically unaccounted for sources of noise in ML pipelines [36]. System level provenance [21] could automatically capture more details about these events and supplement Tribuo provenance to make these changes more visible. Assuming that information is known, users will have to re-create the necessary computational environment themselves before using MERIT so they can achieve full bit-wise reproducibility.

These scenarios highlight an overarching limitation in Tribuo: requiring control over its environment restricts the amount of integration possible with external libraries.

7.3 Guidance for Existing or New ML Libraries

Our experience creating MERIT informed us of two critical and related considerations for ML reproducibility via provenance.

First, provenance is necessary but not sufficient for reproducibility. For example, consider the running example from Section 5.5. A provenance record indicating that Tribuo trained an ensemble model with a non-contiguous set of RNG states is an accurate and valid record. It does not mean a user can easily reproduce the model. MERIT can only reproduce `BaggingTrainers` in the presence of parallelism due to modifications we made to the main Tribuo code. Those modifications also relied on another new feature we added to the library, the ability to set RNG state. Even assuming Tribuo always recorded accurate and valid provenance, without these changes, reproducibility is challenging or impossible.

Second, a provenance collection system benefits from use by a machine driven application such as reproducibility. Otherwise, manual processes can fill in missing details that the provenance system should collect, e.g. the appropriate software version or compute environment. Specifically we identified two benefits: completeness and correctness. In the case of Tribuo, MERIT can verify provenance by reproducing the model it describes. This behavior demonstrates the provenance is correct and complete; Tribuo accurately recorded everything needed to describe the model. Auditors can thereby employ MERIT for verification when using Tribuo provenance as a means of regulatory compliance. When MERIT cannot reproduce a model, as we discussed earlier, we now have a controlled environment in which to debug the issue. Therefore, we recommend that researchers use a specific machine application such as automatic reproducibility or LIMA's memoization to derive the requirements of a provenance collection system.

This approach can apply to existing ML libraries, although having a strong focus on provenance and reproducibility from the beginning changes the way developers will approach the architecture of the library. Some potential changes that might otherwise be added without more thought might seem less attractive as they will affect the overall reproducibility of the library. Systems such as scikit-learn will find using our design difficult, as its authors implemented drastically different design philosophies that focus on making integration with external tools easy [26]. Another challenge comes from systems that let users control the training loop, such as PyTorch [25]. A provenance system can record that it used an external library or integrated user-written code, but it will not necessarily know how to use those artifacts. Existing ML libraries that are already integrated with external libraries have to find a way to integrate provenance throughout the whole system, use the tools described in Section 2, or perform significant refactoring to move towards a system with simpler introspection capabilities.

8 CONCLUSION

We presented MERIT, a reproducibility framework for the Tribuo ML library, enabled by Tribuo's integrated provenance architecture. The framework uses Tribuo model provenance to automatically produce a training pipeline that reproduces the input model. Through the development of MERIT we also improved Tribuo's provenance collection, Tribuo's training implementations, and the reproducibility of external libraries. The framework also produces a diff between the reproduced model provenance and the input model provenance, highlighting differences such as changes in the training data, e.g., a different number of examples or features. This comparison increases confidence in Tribuo's reproduction and provenance collection, making it easier to perform reproducible research in data science and perform audits of ML models. We also identified another use case for the combined Tribuo provenance and reproducibility framework – development of an experimental framework. Since the reproducibility framework controls the variables used to train a model, it is relatively straightforward to assist with model evolution similar to some of the related work we discussed in Section 2. We plan to extend MERIT to support such a framework using the model provenance as a means to derive and train new models in a controlled environment, allowing easy comparisons between different experimental setups and parameter configurations.

[8] https://github.com/tensorflow/tensorflow/issues/48855

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Michele Alberti, Vinaychandran Pondenkandath, Marcel Würsch, Rolf Ingold, and Marcus Liwicki. 2018. DeepDIVA: a highly-functional python framework for reproducible experiments. In *2018 16th International Conference on Frontiers in Handwriting Recognition (ICFHR)*. IEEE, 423–428.

[3] Monya Baker. 2016. 1,500 scientists lift the lid on reproducibility. *Nature* 533, 7604 (2016), 452–454. https://doi.org/10.1038/533452a

[4] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf

[5] Chih—chung Chang and Chih-jen Lin. 2001. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST), 2011* 2, 3 (2001), 280–292.

[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 785–794.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[8] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, 2085–2088. https://doi.org/10.1145/2882903.2899401

[9] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. 2009. Modeling wine preferences by data mining from physicochemical properties. *Decision support systems* 47, 4 (2009), 547–553.

[10] André Elisseeff and Jason Weston. 2001. A kernel method for multi-labelled classification. *Advances in neural information processing systems* 14 (2001), 681–687.

[11] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *the Journal of machine Learning research* 9 (2008), 1871–1874.

[12] R.A. Fisher. 1988. Iris. UCI Machine Learning Repository.

[13] James Gosling, Bill Joy, Guy L Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional.

[14] Odd Erik Gundersen and Sigbjørn Kjensmo. 2018. State of the art: Reproducibility in artificial intelligence. In *Thirty-second AAAI conference on artificial intelligence*.

[15] Philip J Guo and Dawson Engler. 2011. Using automatic persistent memoization to facilitate data analysis scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 287–297.

[16] Matthew Hartley and Tjelvar SG Olsson. 2020. dtoolai: Reproducibility for deep learning. *Patterns* 1, 5 (2020), 100073.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[18] B.S. Lerner and E.R. Boose. 2014. RDataTracker: Collecting Provenance in an Interactive Scripting Environment. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*. https://doi.org/10.1007/978-3-319-16462-5_36

[19] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[20] Tom Mitchell. 1997. Machine Learning. (1997).

[21] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor. 2009. Layering in Provenance Systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (San Diego, California) *(USENIX'09)*. USENIX Association, USA, 10.

[22] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. 2015. noWorkflow: Capturing and Analyzing Provenance of Scripts. In *Provenance and Annotation of Data and Processes*, Bertram Ludäscher and Beth Plale (Eds.). Springer International Publishing, 71–83.

[23] Daniel Nüst and Matthias Hinz. 2019. containerit: Generating Dockerfiles for reproducible research with R. *Journal of Open Source Software* 4, 40 (8 2019), 1603. https://doi.org/10.21105/joss.01603

[24] Michela Paganini and Jessica Zosa Forde. 2020. dagger: A Python Framework for Reproducible Machine Learning Experiment Orchestration. *arXiv preprint arXiv:2006.07484* (2020).

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[26] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[27] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *Proceedings of the 2021 International Conference on Management of Data*. 1426–1439.

[28] Adam Pocock. 2021. Tribuo: Machine Learning with Provenance in Java. *arXiv preprint arXiv:2110.03022* (2021).

[29] Project Jupyter. 2020. repo2docker. https://repo2docker.readthedocs.io/en/latest/

[30] Sheeba Samuel and Birgitta König-Ries. 2020. ReproduceMeGit: A Visualization Tool for Analyzing Reproducibility of Jupyter Notebooks. arXiv:2006.12110 [cs.CY]

[31] Sebastian Schelter, Joos-Hendrik Boese, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. 2017. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems Workshop at NIPS*. 27–29.

[32] Guy L Steele Jr, Doug Lea, and Christine H Flood. 2014. Fast splittable pseudorandom number generators. *ACM SIGPLAN Notices* 49, 10 (2014), 453–472.

[33] Victoria Stodden. 2010. The Scientific Method in Practice: Reproducibility in the Computational Sciences. *SSRN Electronic Journal* (2010). https://doi.org/10.2139/ssrn.1550193

[34] Ana Trisovic, Philip Durbin, Tania Schlatter, Gustavo Durand, Sonia Barbosa, Danny Brooke, and Mercè Crosas. 2020. Advancing Computational Reproducibility in the Dataverse Data Repository Platform. In *Proceedings of the 3rd International Workshop on Practical Reproducible Evaluation of Computer Systems* (Stockholm, Sweden) *(P-RECS '20)*. Association for Computing Machinery, New York, NY, USA, 15–20. https://doi.org/10.1145/3391800.3398173

[35] Joseph Wonsil. 2021. *Reproducibility as a service*. Master's thesis. University of British Columbia. https://doi.org/10.14288/1.0398221

[36] Donglin Zhuang, Xingyao Zhang, Shuaiwen Song, and Sara Hooker. 2022. Randomness in neural network training: Characterizing the impact of tooling. *Proceedings of Machine Learning and Systems* 4 (2022), 316–336.

A PRIMITIVE PROVENANCE TYPES

These provenance types only contain their respective named type as a value, and will not contain another provenance object.

- `BooleanProvenance`
- `ByteProvenance`
- `CharProvenance`
- `DateProvenance`
- `DateTimeProvenance`
- `DoubleProvenance`
- `EnumProvenance`
- `FileProvenance`
- `FloatProvenance`
- `HashProvenance`
- `IntProvenance`
- `LongProvenance`
- `ShortProvenance`
- `StringProvenance`
- `TimeProvenance`
- `URLProvenance`

B METHODOLOGY FOR DEBUGGING REPRODUCIBILITY BUGS

MERIT's control over the training environment provides a stable environment for users to debug reproducibility issues. If a user

reproduces a model, they can test that the reproduced model makes the same predictions on a given test set as the original model. In some cases, the weights are available via a getter method, and they can directly compare the float values as well. If the test set predictions or weights differ than the user knows that MERIT failed to provide bit-wise reproducibility. To examine for root causes, they can check the diff of the two models. If there is an architectural or version difference between the two models, that is likely the culprit. If there are no differences, the root cause might be due to untracked system-level differences: backend library changes, JIT compiler optimizations, or something else. In that case, debugging the issue will require a more in-depth understanding of the computational environments than Tribuo can provide.

C EXAMPLE PROVENANCE DIFFS

We have already presented a diff from a successfully reproduced model in Section 5.6; however, we identified other diffs that we believe shows readily helpful information. For more details on these models, the `reproduce-models.ipynb` notebook in our uploaded artifact (found at: https://doi.org/10.5281/zenodo.7987883) contains the code we used to generate these diffs. Each key in the JSON represents some provenance object, with its value being either another provenance object or a primitive. For example, in Listing 3, a model depended on a `trainer` which ran for 15 `epochs`.

Listing 3 shows the diff between two linear SGD models trained using the same `Trainer` but for a different number of epochs. Listing 4 shows the diff between two identical models trained on different architectures. Listing 5 shows the diff between two models trained using the same `Trainer` but different optimizers.

Listing 3: An example diff of two linear SGD models that used different epochs but are otherwise the same. We have removed the timestamps from the diff for clarity.

```
{
  "trainer" : {
    "epochs" : {
      "original" : "10",
      "reproduced" : "15"
    }
  }
}
```

Listing 4: An example diff of models we trained in the same way but on different architectures. We have removed the timestamps from the diff and removed a key which showed that the path to the data is different since they executed on different machines for clarity.

```
{
  "os-arch" : {
    "original" : "amd64",
    "reproduced" : "aarch64"
  },
}
```

Listing 5: An example diff of two linear models that used different optimizers with the same learning rate. We have removed the timestamps and shortened class names for clarity.

```
{"trainer" : {
    "optimiser" : {
      "class-name" : {
        "original" : "...LinearDecaySGD",
        "reproduced" : "...AdaGrad"
      },
      "rho" : {
        "original" : "0.0"
      },
      "useMomentum" : {
        "original" : "NONE"
      },
      "epsilon" : {
        "reproduced" : "1.0E-6"
      },
      "initialValue" : {
        "reproduced" : "0.0"
      }
    }
  }
}
```

D EXPERIMENTAL DETAILS

All experiments were run using the Oracle OpenJDK 17.0.2 version of Java on both x86 and ARM64 platforms using the default JVM parameters for those platforms. The x86 experiments were performed on an Intel Xeon E-2276M processor and an Intel Xeon E5-2407. The AMR64 experiments were performed on a 4-core VM running on an Ampere Altra. All machines were running Linux.

E ARM64 AND X86 REPRODUCIBILITY

As mentioned in Section 6.2 we reproduced models originally trained on an Intel x86_64 machine on a 4-core ARM64 VM running on an Ampere Altra chip. These models performed the exact same computations as the models trained on x86_64, but in some cases produced slightly different models. Many models were bitwise exact reproductions, e.g. tree based models, however ones which used trancendental functions from `java.lang.Math` exhibited slight differences. Specifically, we had replication issues with classification models which used a softmax or RBF function, such as LibSVM with RBF, Logistic Regression, and Factorization Machines. These models depend upon repeated applications of Math.exp and Math.log which increase the divergence between ARM & x86's implementations. In Java 16 (also checked on Java 8 and Java 11), the aarch64 platform (i.e., ARM64) uses `java.lang.StrictMath` to provide `java.lang.Math`, while on the x86_64 platform there are faster implementations of several `java.lang.Math` functions which have slightly looser bounds (they are allowed a 2 ulp difference from the correctly rounded floating point value[9]). With

[9]https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Math.html

repeated iterations of the trancendental functions (e.g., via mini-batch stochastic gradient descent) these slight differences can add up to larger differences in the weights, which under certain conditions are visible as causing different predictions on large datasets. In our test datasets we only observed weight differences, all the evaluations remained the same, but in other experiments we have observed slightly different predictions from otherwise identically trained models on x86_64 and ARM64. Additionally we noticed that between Java 8 and Java 16 on x86_64 some of the implementations of `java.lang.Math` were changed and these changes again result in slightly different model weights, which in specific circumstances can result in different predictions.

F EXAMPLE PROVENANCE

We present an example model provenance for a logistic regression model trained on the Iris dataset. It is split into two parts for ease of presentation with Listing 6 presenting the model provenance excluding the data provenance, and Listing 7 presenting the data provenance.

Listing 6: Example model provenance for a logistic regression trained on Irises. The DataSource provenance is presented separately in Figure 7 to improve readability, it would appear in the "source" key.

```
{ "class-name" : "org.tribuo.classification.
    sgd.linear.LinearSGDModel",
    "dataset" : {
      "class-name" : "org.tribuo.
          MutableDataset",
      "datasource" : {
        "class-name" : "org.tribuo.
            evaluation.TrainTestSplitter",
        "is-train" : "true",
        "seed" : "1",
        "size" : "150",
        "source" : <See next figure>,
        "train-proportion" : "0.7" },
      "is-dense" : "true",
      "is-sequence" : "false",
      "num-examples" : "105",
      "num-features" : "4",
      "num-outputs" : "3",
      "transformations" : [ ],
      "tribuo-version" : "4.2.0-SNAPSHOT" },
    "instance-values" : { },
    "java-version" : "16.0.2",
    "os-arch" : "amd64",
    "os-name" : "Linux",
    "trained-at" : "2021-10-01T17
        :46:21.743914022Z",
    "trainer" : {
      "class-name" : "org.tribuo.
          classification.sgd.linear.
          LogisticRegressionTrainer",
      "epochs" : "5",
      "host-short-name" : "Trainer",
      "is-sequence" : "false",
      "loggingInterval" : "1000",
      "minibatchSize" : "1",
      "objective" : {
        "class-name" : "org.tribuo.
            classification.sgd.objectives.
            LogMulticlass",
        "host-short-name" : "LabelObjective"
            },
      "optimiser" : {
        "class-name" : "org.tribuo.math.
            optimisers.AdaGrad",
        "epsilon" : "0.1",
        "host-short-name" : "
            StochasticGradientOptimiser",
        "initialLearningRate" : "1.0",
        "initialValue" : "0.0" },
      "seed" : "12345",
      "shuffle" : "true",
      "train-invocation-count" : "0",
      "tribuo-version" : "4.2.0-SNAPSHOT"},
    "tribuo-version" : "4.2.0-SNAPSHOT" }
```

Listing 7: Example DataSource provenance, pulled from Figure 6 where it goes in the "source" key from the model's provenance. Additionally, due to space concerns we removed three "DoubleFieldProcessors". They are the same as the one listed here but with the following values for "fieldName": "petalWidth", "sepalWidth", and "sepalLength".

```
{   "class-name" : "org.tribuo.data.csv.
    CSVDataSource",
    "dataPath" : "/home/jupyter/results/
        bezdekIris.data",
    "datasource-creation-time" : "2021-10-01
        T17:46:21.449951708Z",
    "file-modified-time" : "1999-12-14T20
        :12:39Z",
    "headers" : [ "sepalLength", "sepalWidth
        ", "petalLength","petalWidth", "
        species" ],
    "host-short-name" : "DataSource",
    "outputFactory" : {
      "class-name" : "org.tribuo.
          classification.LabelFactory"
    },
    "outputRequired" : "true",
    "quote" : "\"",
    "resource-hash" :"0
        FED2A99DB77EC533A62DC66894D3EC6DF3B58
         B6A8F3CF4A6B47E4086B7F97DC",
```

```
"rowProcessor" : {
  "class-name" : "org.tribuo.data.
      columnar.RowProcessor",
  "featureProcessors" : [ ],
  "fieldProcessorList" : [ {
    "class-name" : "org.tribuo.data.
        columnar.processors.field.
        DoubleFieldProcessor",
    "fieldName" : "petalLength",
    "host-short-name" : "FieldProcessor
        ",
    "onlyFieldName" : "true",
    "throwOnInvalid" : "true"
  }, ... ],
  "host-short-name" : "RowProcessor",
  "metadataExtractors" : [ ],
  "regexMappingProcessors" : { },
  "replaceNewlinesWithSpaces" : "true",
  "responseProcessor" : {
    "class-name" : "org.tribuo.data.
        columnar.processors.response.
        FieldResponseProcessor",
    "defaultValues" : [ "" ],
    "displayField" : "false",
    "fieldNames" : [ "species" ],
    "host-short-name" : "
        ResponseProcessor",
    "outputFactory" : {
      "class-name" : "org.tribuo.
          classification.LabelFactory"
    },
    "uppercase" : "false"
  },
  "weightExtractor" : {
    "class-name" : "org.tribuo.data.
        columnar.FieldExtractor"
  }
},
"separator" : ","
}
```